# Warzone Architecture Design - Build 2

This document is an extension to the architecture defined in build 1 in that it further defines how the application manages the different games phases (State pattern), how it manages processing of messages from different parts of the application (Observer pattern), and how it manages the input and processing of the different orders (Command pattern).

The architecture design document for build 1 defined the overall application structure, i.e. underlying MVC pattern. See the Architecture Design from build 1 for a high-level description.

## Message Processing – Observer Pattern



Figure - Message Processing Diagram

Figure 1, shows a package diagram with the relevant classes that makeup how messages are processed throughout the application so that 1) users can receive messages from anywhere in the application, 2) the application is testable, 3) the MVC architecture is preserved.

The following describes the process illustrated in the diagram:

* Views are responsible for displaying information/messages to the user, and for accepting user input.
* Views that attach to LogEntryBuffer get notified whenever a new message has been set, and detach when it no longer needs to receive notifications of new messages.
* Messages can be set from any Model or Controller based classes. Message should only include those that need to be seen by the user.
* Messages that are displayed to the user must also be logged to a log file.
* With reference to the Observer pattern, the ‘Subject’ is the LogEntryBuffer and the ‘Observer’ are the view classes.

## Game Phase Processing – State Pattern



Figure 2 – Game Phase Processing Diagram

Figure 1, shows a package diagram with the relevant classes that makeup how the different game phases are processed throughout the application. The idea is to use the State behavioural pattern so that 1) makes it relatively easy and dynamic to modify the logic dictating the rules between phases, 2) the application makes the application more testable as phase logic is kept closer to the logic that needs to decide the next phase, and that it’s not imbedded in big complex methods.

The following describes the process illustrated in the diagram:

* The GameEngine class invokes the different phases vie the execPhase() method based on an attribute that holds the current/next phase.
* Phase classes inherit from the abstract Phase class, and invoke the nextPhase() method to set the next phase in the GameEngine.
* IssueOrder, OrderExec, and Reinforcement classes are sub-phases of the Gameplay phase. They inherit extra functionality from the GamePlay phase as they are part of that phase.
* With reference to the State pattern, the ‘Context’ is the GameEngine and the ‘State’ classes are the controller based phase classes.
* The following diagram illustrates the game flow between phases:



Figure - Game Phase State Diagram

## Game Player Order Processing – Command Pattern



Figure 4 – Game Player Order Processing Diagram

Figure 1, shows a package diagram with the relevant classes that makeup how the different game processes player orders during gameplay. The idea is to use the Command behavioural pattern so that 1) makes it relatively easy and dynamic to add/modify the logic dictating the processing of orders, 2) orders can be created and executed at different times during gameplay 3) the application makes the application more testable as order logic is kept abstract and consistent.

The following describes the process illustrated in the diagram:

* The IssueOrderController is invoked which sets up the environment and invokes the issue\_order() method for each player.
* The issue\_order() (for a human player) invokes the getOrder() method defined in the as a IGameplayOrderDatasource. Currently the IssueOrderController is defined as the datasource to get the order command from the user. This is setup as it is to respect the MVC pattern, i.e. the controller gets user input from the view.
* Once all the orders have been created in the IssueOrderController, the game engine will invoke the OrderExecPhase execPhase() method (as defined by the game phases in the previous section). This is where the orders are executed in round robin fashion.

## Player Behavior – Strategy Pattern

Player behaviors dictate how a player will decide to play a game. In the case of a human player the system receives commands from human interaction. Those commands are translated into orders. In the case of computer based players each respective player strategy class will decide on different orders based on the strategy of that player. The type of strategy (behavior) for each player is established during player creation.



Figure 5 - Player Behavior Diagram

The following describes the process illustrated in the diagram:

* Player strategies (setStrategy) are set when either issuing the ‘tournament’ or ‘gameplayer’ commands.
* The Player class invokes the strategy object’s createOrder() method during the issue\_orders phase of the game.
* Each player strategy class uses a different set of heuristics to decide on the next order

## Conquest Maps – Adaptor Pattern

This architecture which makes use of the Adaptor pattern allows the game the ability to load and save maps in either the original Domination style map or the new Conquest style maps. Both style of maps support different formats, but essentially contain the same map information.



Figure 5 - Player Behavior Diagram

The following describes the process illustrated in the diagram:

* Different controllers (depending on the phase of the game) invoke the getMapIo() method.
* When loading a map, getMapIo() calls functions that analyze the map file to decide if the map to load is a Domination style map or a Conquest style map. Depending on the format getMapIo() instantiates either the MapIoDomination class or MapIoAdaptor for Conquest style maps. MapIoAdaptor translates the corresponding MapIoDomination calls into calls to MapIoConquest to get the conquest map data, and then translates the map information into the same format produced by MapIoDomination.
* When saving a map, the system indicates if it requires it to save as a Domination style map or a Conquest style map. Then based on that getMapIo() instantiates either the MapIoDomination class or the MapIoAdaptor.   
  MapIoAdaptor then translates the corresponding MapIoDomination calls into calls to MapIoConquest to save the conquest map.